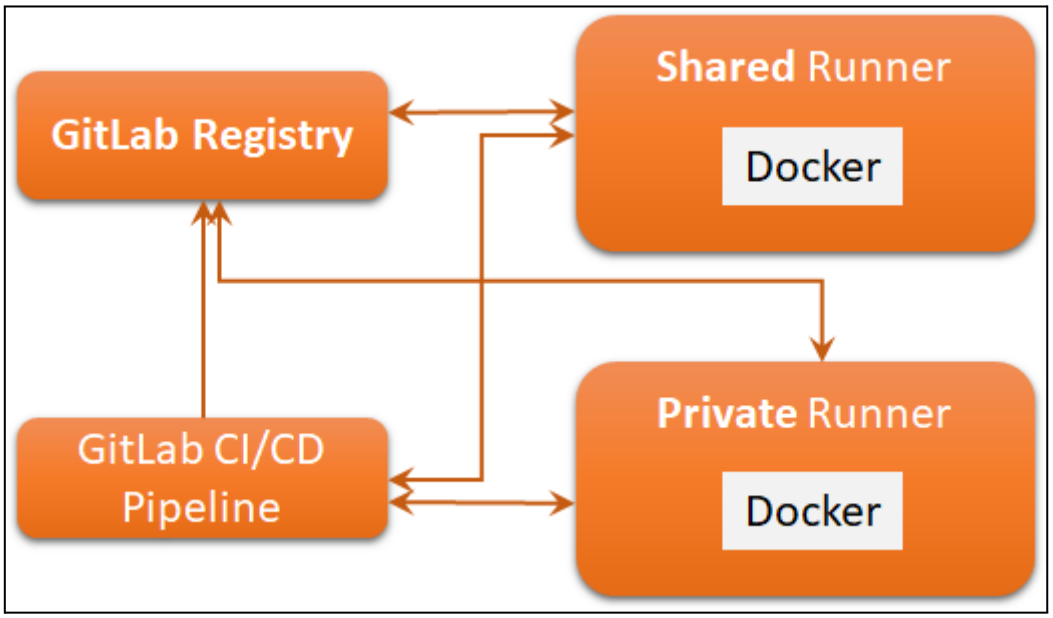
Practice Session-07:- CI/CD with GitLab

Make sure that you have already gone through Lab-06.

In this session you're going to learn the basic building blocks of the GitLab CI/CD pipeline and create the CI/CD pipelines to automatically deploy your custom code on the docker containers.



Understanding the communication among GitLab CICD, runneres and the registry.

Prerequisite

- Basic knowledge on Python, pandas library, csv file format
- Basic knowledge on YAML
- Familiar with Dockerfile
- Familiar with Git

Exercise 1: Setting up of gitlab project and runners

In this exercise you will create a project and set up GitLab Runner and configure the runner, configure pipeline, etc. In this lab we are going to use the flask application code from Lab 02 and k8s deployment files from Lab 04.

- Create a project in the gitlab with name `lab07-gitlab-cicd` under group `Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>`
- Clone the project to `k8s-controller` VM and go to the project `cd lab07-gitlab-cicd`

- Create a directory `webapp-flask` and add the flask web application code used in Lab 02. It should include following files
    - Dockerfile
    - templates/home.html
    - app.py
    - requirements.txt
- Add the `flask_deployment.yaml` and `pv.yaml` files outside the `webapp-flask` directory. These files were created in Lab 04. (You should be able to get these files from your `Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>/k8s-deployment` project).
- Commit with the message "`Added the required code and deployment files`" and push the code.

Installation and registration of Gitlab runners

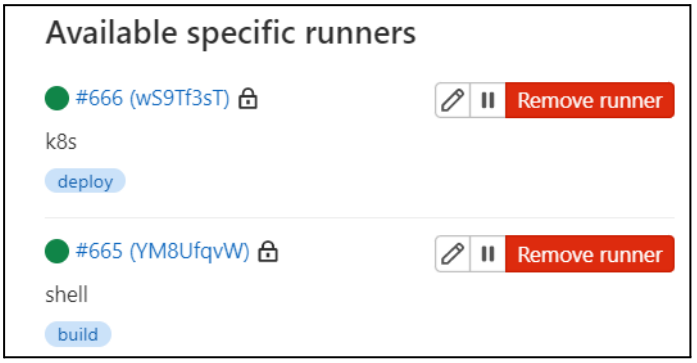
In this task, we are going to set up two runners with `shell` and `docker` as executors. The `shell` executor based runner used for building the docker images and `docker` executor based runner used to deploy the application on the kubernetes cluster.

- Make sure that you are logged in to your `k8s-controller` VM.
- A gitlab runner (with `docker` executor) in your `k8s-controller` VM should be ***running***. You installed it in Lab06, Exercise 05. You can check the status using the command `sudo gitlab-runner list`
    - You may reuse that runner in this Practice session. For this you need to follow some additional steps (DIY). In this case, you don't have to register the second runner as given in the table below.
    - If you don't want to reuse that runner, you may uninstall that runner. Find the exact command : `sudo gitlab-runner unregister --help`. In this case you need to add the second runner as well given in the below table.
- Get the token for Gitlab runner registration
    - Go to your GitLab project, should be available at `Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>/lab07-gitlab-cicd`
- Go to `Setting` --> `CI/CD` → Expand `Runner`
- Note down `Registration token`
- Now let's register the below gitlab runner(s) to your project.

	First runner	**Second runner**
GitLab instance URL	https://gitlab.cs.ut.ee/	https://gitlab.cs.ut.ee/

Registration token	as noted in the earlier step	as noted in the earlier step
Description for the runner	As per your convenience	As per your convenience
Tag	`build`	`deploy`
Executor	`shell`	`docker`
Default docker image	- Not applicable -	`ubuntu:18.04`

- Now, you should see the both the runners in running state as shown below



- Create k8s agent for this project, this is similar to the task performed in [Lab 04](#) Exercise 4 , Task 4.2
    - This task should be performed in the Master (node1) node of the k8s cluster
    - Give agent name as `k8s`
- We need to create an access token for authentication when reading and writing to the GitLab container registry. This access token is required in later exercise, and please note it down carefully (Note!! Please copy the access token and save it in a text file at some place)
    - Go to Settings→Repository→Depoy tokens and Expand it.
        - Name: k8s
        - Expiration date: Choose some date
        - Username (optional): Your Gitlab Username
        - Scopes (select at least one):read_registry,write_registry
        - Click on Create deploy token
        - Copy the access token and save it in a text file someplace.

Exercise 2: Building your first pipeline

In this exercise, you're going to create the `.gitlab-ci.yml` CI file (if not present), a YAML file containing a specific set of jobs, stages, tags, etc., for the GitLab CI/CD pipeline. For information on the keywords used in this CI file can be found at
https://docs.gitlab.com/ee/ci/yaml/
In this file, you define

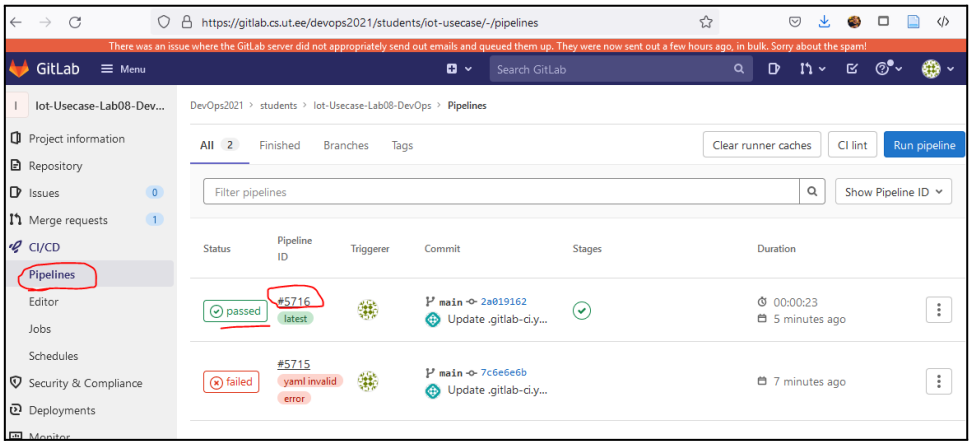
- The structure and order of jobs that the runner should execute.
- The decisions the runner should make when specific conditions are encountered.

At this point, you already have registered two GitLab runners.

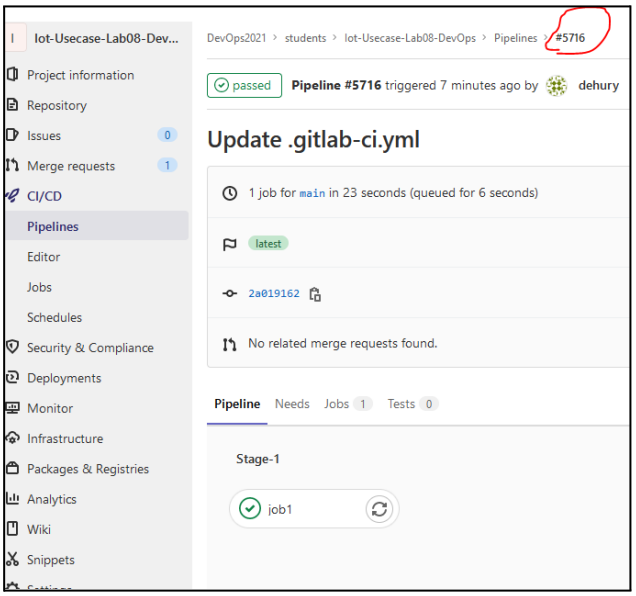
Points to Remember,

1. In this exercise, you will very frequently commit the changes and push the code to your repo (you may use Gitlab Web IDE to modify the files).
2. At the end of this exercise, we may go through the commits history. So make sure that you have followed and implemented each step one after another.

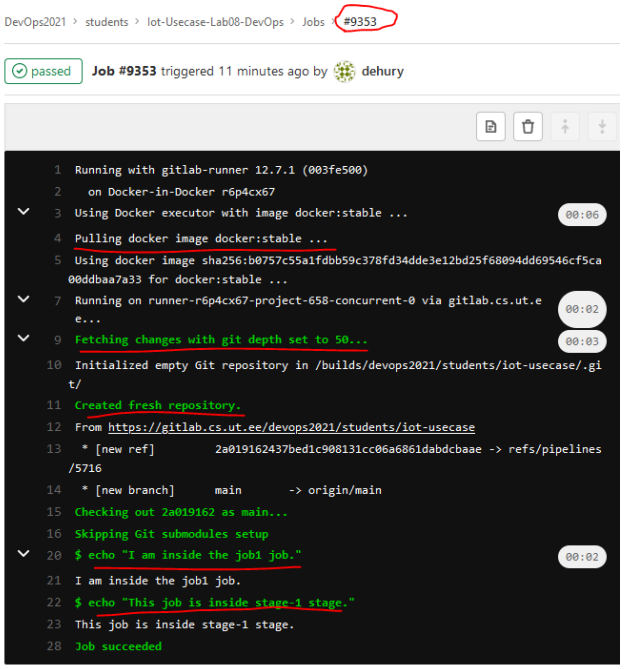
- After each commit, you can see pipeline status in `CI/CD` → `Pipelines`, as below



- Click on the Pipeline ID to see the list of Jobs. for example, when I click on Pipeline ID `#5716`, I see the following list and status of jobs:



- Now, if you click on the job (e.g. `job1` in the above figure), you can see the logs of gitlab runner. For example, when I click on the above job1, I see the following output:



Note!! You will commit and push the code several times in further exercies.

Remember that, we may see the commit history while grading your submission. So, please specify the commit messages as prescribed.
Let's move ahead and prepare our first pipeline.

2.1: Single stage pipeline

- Create `.gitlab-ci.yml` CI file (You may use GitLab UI to create this file) and add only one stage `stage-1` as below:

```
stages:
  - stage-1

job1:
  stage: stage-1
  script:
    - echo "I am inside the job1 job."
    - echo "This job is inside stage-1 stage."
```

- Here the first and only job is `job1`, which will run in `stage-1` stage.
- Commit with the message “Added job-1 in stage-1” and push the above changes.
- See the newly created pipeline and job.

2.2 : Lets print some predefined and custom variables

- To modify the `.gitlab-ci.yml` in further steps, you may use Pipeline Editor in GitLab UI.

- Modify the `script` section and `echo` the following predefined variables:
  - CI_JOB_STAGE
  - CI_COMMIT_BRANCH
  - CI_COMMIT_AUTHOR
  - CI_COMMIT_DESCRIPTION
  - CI_COMMIT_MESSAGE
  - CI_CONFIG_PATH
  - CI_JOB_NAME
  - CI_JOB_ID
  - CI_JOB_STATUS
  - CI_PIPELINE_ID
  - CI_RUNNER_ID
- To print a predefined variable, you can use the following command in `script` section:
  - `echo $<variable_name>`
  - E.g. `echo $CI_JOB_STAGE`
  - Commit with the message “Printing predefined variables” and push the above changes.
- To see all the available variables, you can use `- export` OR `- env` in the script section.

```
script:
  - echo "I am inside the job1 job."
  - echo "This job is inside stage-1 stage."
  - export
```

- Define your own variables in the configuration file as below. This variable is accessible to all jobs. Create the `IMAGE_HUB` as your custom variable (sample given in the below figure)

```
variables:
    IMAGE_HUB: gitlab.cs.ut.ee:5050/poojara/

stages:
    - stage-1
job1:
    stage: stage-1
    script:
        - echo "I am in job"
        - echo "This job is job1"
        - echo $IMAGE_HUB
```

- Add an `echo` statement to print the above `IMAGE_HUB` custom variable.

- Commit with the message “Printing the user defined variables inside .gitlab-ci.yml” and push the above changes.
- The other way to define variables is by the use of the `Variables` feature in GitLab. The `Variables` feature can be found in `settings` → `CI/CD` → Expand `Variables`. These variables can be used in other pipelines as well.
    - Add a variable ‘`my_project_wide_variable`’ with value `<your_name>` in `settings` → `CI/CD` → Expand `Variables`.
    - `echo` the variable in the pipeline under the `script` section.
- Create a variable to store the GitLab acces token using Variable feature
    - Add a variable `gitlabpassword` with value of access token created in `Exercise 1`
- Commit with the message “Printing the user variables” and push the above changes.

2.3 : Let the pipeline run on a specific gitlab-runner.

- At this moment, you have already registered your specific gitlab-runner with the tags `build` and `deploy`.
- You will use the above tags to run your jobs in your runner in the **k8s-controller** VM.
- Modify the configuration file again, so that all the jobs will run in your specific runner.

```
job1:
  stage: stage-1
  tags:
    - build
```

- Commit with the message “Added the tags” and push the above changes.
- See the newly created pipeline and job.

2.4: Working with GitLab job artifacts, “before_script”, “script”, and “after_script”

- Artifacts are used to specify which files to save as job artifacts. Jobs can output an archive of files and directories. This output is known as a job artifact.
    - Artifacts can be mentioned as shown below. Update your `.gitlab-ci.yml` file as shown below

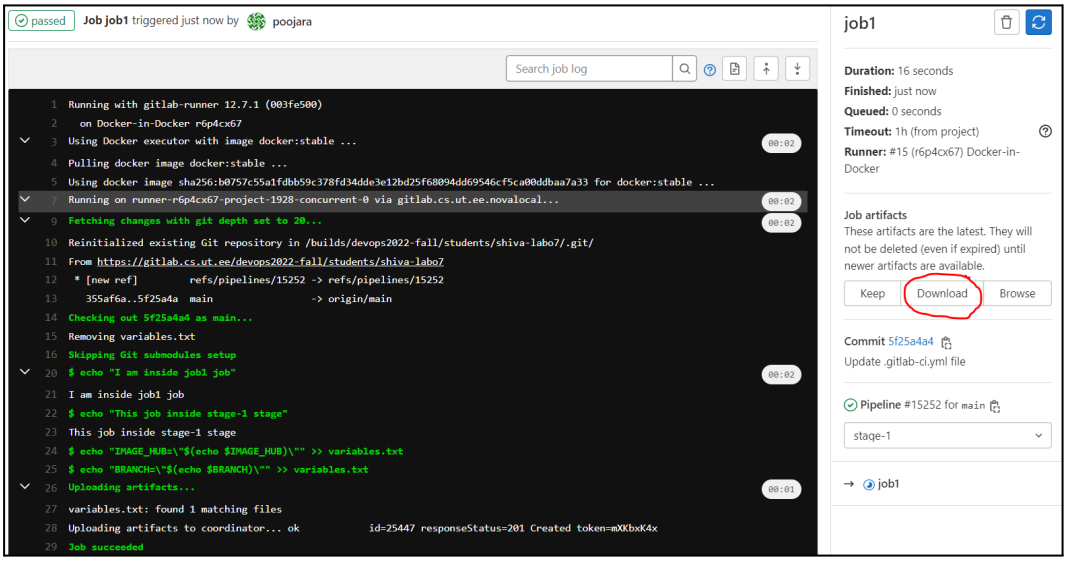
```
variables:
    IMAGE_HUB: gitlab.cs.ut.ee:5050/poojara/

stages:
    - stage-1
job1:
    stage: stage-1
    script:
        - echo "I am inside job1 job"
        - echo "This job inside stage-1 stage"
        - echo "IMAGE_HUB=\"$(echo $IMAGE_HUB)\"" >> variables.txt
        - echo "BRANCH=\"$(echo $BRANCH)\"" >> variables.txt

    artifacts:
        paths:
        - variables.txt
```

    - Commit with the message “Added artifacts for job1” and push the above changes.

- You can download your job artifacts, after the pipeline is executed



- The `before_script`, and `after_scrip` are used to define an array of commands that should run before and after all the jobs under `script` tag,
  - Add the code as shown below

```yaml
variables:
    IMAGE_HUB: gitlab.cs.ut.ee:5050/poojara/

stages:
    - stage-1
job1:
    stage: stage-1
    before_script:
        - echo "Execute this command before any 'script:' commands."

    script:
        - echo "I am inside job1 job"
        - echo "This job inside stage-1 stage"
        - echo "IMAGE_HUB=\"$(echo $IMAGE_HUB)\"" >> variables.txt
        - echo "BRANCH=\"$(echo $BRANCH)\"" >> variables.txt
    after_script:
        - echo "Execute this command after any 'script:' commands."
    artifacts:
        paths:
        - variables.txt
```

  - Commit with the message “Added before and after script references” and push the above changes.

Exercise 3: Working with gitlab container registry

In this task, you modify the flask web application, create its docker image, and push it to the gitlab container registry. Further, you are going to deploy it on k8s cluster.

In this step, we will prepare a flask web application. So at this point, you have a `webapp-flask/Dockerfile` and `webapp-flask/templates/home.html` files.

- Modify an `home.html` file inside `/templates` directory.

- Add (inside <body> tag) the following content to your html file to display the message

  "`Hi I am <your_name> !! This is to demonstrate the gitlab image build`"

- Now, you need to update the `.gitlab-ci.yml` CIfile with two stages: `build` and `run`. At this point, we can remove the previous stage `stage-1`

```
stages:
    - build
    - run
```

- Now let's create two jobs:
  - `build_image` to build the image
  - `run_image` to run the image
- Define `build_image` job in `.gitlab-ci.yml` file
  This job should run in the `build` stage.
  Under `script`:
  - Login to gitlab. Make sure that the `gitlabpassword` variable is defined in your Gitlab project.
  - Create a `IMAGE_NAME` variable with the value `gitlab.cs.ut.ee:5050/<name_space>/<project_name>/<image_name>:<tag_name>`
  - Build the docker image using `Dockerfile` present in the project directory.
  - Push the docker image to the container registry.

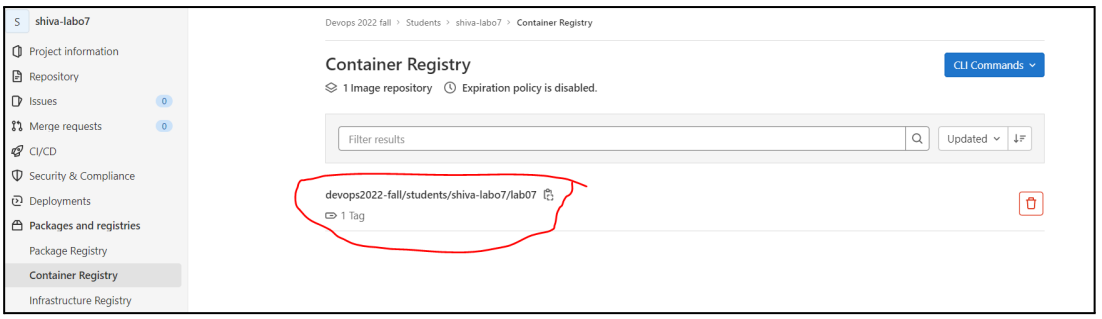
The update `.gitlab-ci.yml` file should look similar to below. Below image is just for reference purpose.

```
variables:
    IMAGE_HUB: gitlab.cs.ut.ee:5050/poojara/
    IMAGE_NAME: gitlab.cs.ut.ee:5050/devops2022-fall/all-solutions/lab07-gitlab-cicd/lab07

stages:
    - build
    - run

build_image:
    stage: build
    script:
        - docker login -u poojara -p $gitlabpassword gitlab.cs.ut.ee:5050
        - docker build -t $IMAGE_NAME -f ./Dockerfile .
        - docker push $IMAGE_NAME
    tags:
        - build
```

  - Commit with the message "Added build_image job" and push the above changes.
  - Once the pipeline is executed, you should see the container image in the gitlab registry.



- Defining `run_image` job in `.gitlab-ci.yml` file (This is similar to .gitlab-ci.yml file from [Lab04](Lab04))
  This job should run in `run` stage
  - Add the `image` with `name: bitnami/kubectl:latest` and entrypoint: `[""]`

  Under `script`:
  - Under script, add the kubectl config setting commands (This is similar to .gitlab-ci.yml file from Lab04)
  - Add the `kubectl config use-context Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>/lab07-gitlab-cicd:k8s`

  - Add the command to create the kubectl secrets. This is used to read and pull the docker image stored in the gitlab container registry (Here, `lab07` container image that you pushed into the gitlab registry in the previous step). In the below command `REGISTRY_USERNAME` should be gitlab username and `REGISTRY_PASSWORD` should be `$gitlabpassword`
    `kubectl create secret docker-registry registry-credentials --docker-server=https://gitlab.cs.ut.ee:5050 --docker-username=REGISTRY_USERNAME --docker-password=REGISTRY_PASSWORD --docker-email=REGISTRY_EMAIL || true`

The updated `.gitlab-ci.yml` file after defining `run_image` job should look like below:

```
24  run_image:
25      stage: run
26      image:
27          name: bitnami/kubectl:latest
28          entrypoint: [""]
29      script:
30          - kubectl config get-contexts
31          - kubectl config use-context devops2022-fall/students/shiva-labo7:k8s
32          - kubectl config current-context
33       #   - kubectl delete secrets registry-credentials
34          - kubectl create secret docker-registry registry-credentials --docker-server=https://gitlab.cs.ut.ee:5050 --docker-username=poojara
    --docker-password=$gitlabpassword --docker-email=poojara@ut.ee || true
35          - kubectl get pods
36          - kubectl delete -f ./flask_deployment.yaml || true
37          - kubectl delete -f ./pv.yaml || true
38          - kubectl apply -f ./pv.yaml
39          - kubectl apply -f ./flask_deployment.yaml
40          - kubectl get svc
41          - kubectl get po
42      tags:
43          - deploy
44
```

- Modify the `flask_deployment.yml` at line # 16 `image` with value of `$IMAGE_NAME`
  Ex: `image: gitlab.cs.ut.ee:5050/devops2022-fall/students/shiva-labo7/lab07`

- Allow the pod (you need to update flask_deployment.yml) to use your secret (`registry-credentials`) while pulling the private docker image from your gitlab container registry (use `imagePullSecrets`).
- Commit with message “added the run_image stage”and **push** above changes.
- See the newly created pipeline and jobs.
- Login and check the deployment running in master (node1) node using command `kubectl get po -o wide`
- Get the nodePort address under service `kubectl get svc`
- At the end, you should be able to see the web page at http://MASTER_NODE_EXT_IP:NODEPORT_ADDRESS

Screenshot - 1

`Take a screenshot of a webpage and IP address are clearly seen.`

Exercise 4: Working with image build versioning and updating the application

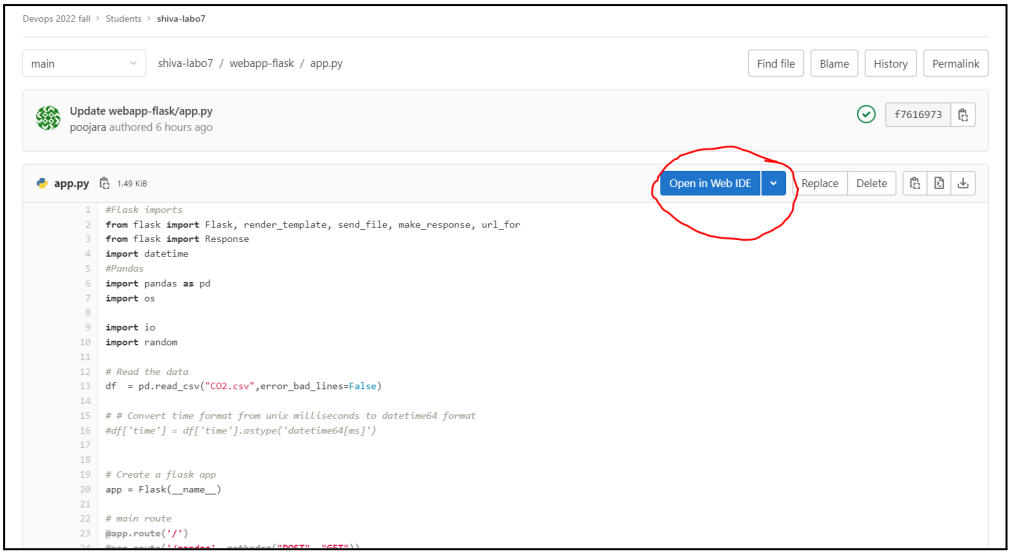
In this task, your going to update the flask web application to display the minimum and maximum CO2 values on the web page. Further, you learn about container image versioning.Its not good practice to tag the images always with tag “latest” for every pipeline execution. This is because, you may run in to the following problems

1. If you re-execute an older CI job (or if you run the same CI job in multiple testing / feature Git branches), the CI jobs will keep overwriting the latest tag. “latest” loses its meaning. Your production environment will most likely become unstable if you configure it to use the latest tag of your image.
2. It would become impossible to use some older version of your image on purpose in some of your deployments.
   To overcome this issue, you can use or tag the build version using GitLab CI/CD environment variables mentioned below:

Git tag	`CI_COMMIT_TAG`
Git commit SHA-256 hash	`CI_COMMIT_SHA`
Shortened Git commit hash	`CI_COMMIT_SHORT_SHA`
Git branch name	`CI_COMMIT_BRANCH`
date + timestamp	`CI_JOB_STARTED_AT`
unique build number	`CI_JOB_ID`

4.1 : Modify the flask application

Now, let us modify the flask application code under directory `webapp-flask` in the `gitlabpoject` and to modify/edit the code,you may use the gitlab Web IDE.



- Let us modify the app.py to process the `CO2.csv` and add the code snippet should be added under `def GK():`
    - Comment the line # 16 (df['time'] = df['time'].astype('datetime64[ms]'))
    - Get the minimum and maximum timestamp from the `time` column to get the duration of observation.
    - Convert the timestamp to human readable format.
    - Read the values of the `value` column. Find min and max values using the `pandas` library.
    - Add the duration, min value, and max value to the `home.html` file present in `/templates/home.html` directory.

      The final code look like

```
from flask import Response
import datetime
#Pandas
import pandas as pd
import os

import io
import random

# Read the data
df  = pd.read_csv("CO2.csv",error_bad_lines=False)

# # Convert time format from unix milliseconds to datetime64 format
#df['time'] = df['time'].astype('datetime64[ms]')


# Create a flask app
app = Flask(__name__)

# main route
@app.route('/')
@app.route('/pandas', methods=("POST", "GET"))
def GK():
    # Min time

    tmp = df['time'].min()
    min_date = datetime.datetime.fromtimestamp((tmp/1000000000.0)).strftime('%Y-%m-%d %H:%M:%S.%f')
    # Max time
    tmp = df['time'].max()
    max_date =  datetime.datetime.fromtimestamp((tmp/1000000000.0)).strftime('%Y-%m-%d %H:%M:%S.%f')

    min_val = df['value'].min()
    max_val = df['value'].max()

    cwd = os.getcwd()
    index = open(cwd + '/templates/home.html',"a")

    index.write("<br/>")
    index.write("The duration of data is [  "+str(min_date)+" ~ "+str(max_date)+"  ]<br/>")
    index.write("The MIN CO2 value observed is:"+str(min_val)+"<br/>")
    index.write("The MAX CO2 value observed is:"+str(max_val)+"<br/>")

    index.close()
    return render_template('home.html',PageTitle="plot")
    #return render_template('home.html',
   #                        PageTitle = "plot",table=[df.to_html(classes='data', index = False)], titles=df.columns.values)

if __name__ == '__main__':
    app.run(debug = True,host='0.0.0.0',port=5000)
```

- Now, modify the `templates/home.html` and here, comment the following code block

```
{% for table in table %}
{{ table|safe }}
{% endfor %}
```

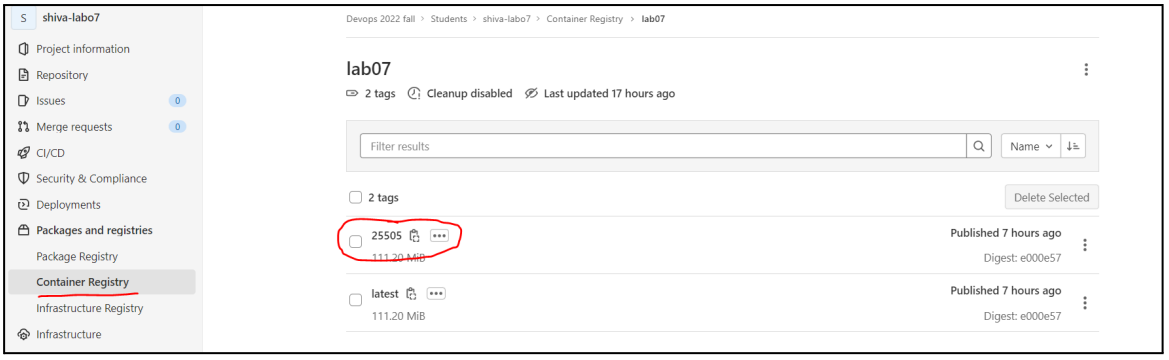
4.2 : Edit .gitlab-ci.yml file

Here basically you will update `.gitlab-ci.yml` to tag a container image while building.

- Update `IMAGE_NAME:` as `gitlab.cs.ut.ee:5050/devops2022-fall/students/shiva-labo7/lab07:$CI_JOB_ID`
    - You may use different variables as image tag, e.g. $CI_COMMIT_SHORT_SHA
- You can use any of the gitlab environment variables to tag the image as mentioned in the introduction of Exercise 4.
- Commit the project with message as “Updated the app.py and home.html to display min, max CO2 values”

4.3 : Checking the output of the flask application and container registry

- Once, you commit in 4.2 the pipeline is executed and you can see the *build_image* job output in container registry as shown below



- You can check flask application running by using Master (node1) external IP address and NodePort address
- The final flask application output is displayed as:

Screenshot - 2

Take a screenshot of a webpage and IP address are clearly seen.

Deliverables

1- Gather all the screenshots

- Screenshot 1
- Screenshot 2

2- Download code of your GitLab project

3- Zip the code, screenshot and Upload the zip file to the course wiki page.

4- You may **Stop** the Virtual Machines and you can start using the same in the next **practice session.**

Don't delete your VMs